UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Sistema de planificación de proyectos de videovigilancia

Borja González Farías
Tutor: Eduardo Cermeño Mediavilla
Ponente: Juan Alberto Sigüenza Pizarro
Junio 2019

SISTEMA DE PLANIFICACIÓN DE PROYECTOS DE VIDEOVIGILANCIA

AUTOR: Borja González Farías
TUTOR: Eduardo Cermeño Mediavilla

Escuela Politécnica Superior
Universidad Autónoma de Madrid
junio de 2019

Resumen

El proyecto desarrollado trata de un sistema de planificación de proyectos de videovigilancia. Hay dos usuarios: el **común** y el **administrador**.
El usuario administrador es el que puede crear, modificar y eliminar cámaras; cada cámara es creada con 4 datos: el nombre, la distancia focal, un tamaño de sensor (en pulgadas) y una resolución. El administrador en cualquier momento es capaz de eliminar o modificar cualquiera de las cámaras que haya creado.
El usuario común, por su lado, tiene la funcionalidad en la que se centra este proyecto. El usuario dispone de un mapa en donde podrá seleccionar el punto en donde desea colocar una cámara (antes seleccionada), y según las características de la cámara y el objeto que se quiera detectar (antes seleccionado y habiendo pulsado el botón de "Aceptar"), ésta mostrará, gráficamente, un campo de visión del área que es capaz de "ver".
Una vez que haya cámaras en el mapa, el usuario podrá girarlas a su gusto a través de un formulario a la derecha en donde introduce la cantidad de grados (número entero) que quiere aumentar a la cámara respecto a su centro (si la cantidad es positiva, gira en sentido de las manecillas del reloj, y si es negativa, gira en sentido contrario); habiendo introducido la cantidad, se pulsa en el botón "Rotar" y se hace click a la cámara que se desea girar.
Ya que el usuario haya puesto las cámaras que haya deseado, podrá exportar ese proyecto a una imagen PNG o JPEG, pulsando en el botón "Exportar"; aparecerá una ventana con el explorador de archivos para seleccionar la ubicación donde el usuario desee guardar el proyecto, selecciona el formato de la imagen que quiera con la 'combobox' disponible y finalmente pulsa en "Exportar".

Palabras clave

- Cámara
- Campo Visual
- Mapa

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Contexto

Según el Ministerio del Interior, en la primera mitad del 2016, se produjeron 16.342 robos en la Comunidad de Madrid, 333 más que en 2015 en el mismo periodo[6]. Esto ha provocado que sean cada vez más los diferentes establecimientos (viviendas, negocios, etc.) que apuestan por una seguridad activa instalando cámaras de seguridad.

Según un estudio de HKExnews de 2015, el mercado mundial de la videovigilancia ha crecido enormemente pasando de generar 5,6 MDD (millones de dólares) en 2009 a generar 11,5 MDD en 2014, y se estimó que para 2019 los ingresos serían de 22,3 MDD[5]. Todo esto indica que el mercado de la videovigilancia está en auge y lo seguirá estando por bastante tiempo más.

A pesar de todo este aumento de la videovigilancia, existe un problema, y es que muchas cámaras de seguridad generan cantidades exorbitantes de vídeo, y es por esto que muchas veces, por falta de tiempo o recursos, las grabaciones se quedan sin revisar y varios incidentes de seguridad, como intrusiones de personas no deseadas al recinto o cualquier otra actividad sospechosa, pasan completamente inadvertidos. Es por esto que la analítica de vídeo se ha introducido al mundo de la seguridad; ésta hace referencia a aplicaciones software que generan metadatos para enumerar diferentes objetos detectados como personas, coches, furgonetas, etc., y en base a estos datos se pueden iniciar diversas acciones (si enviar una alerta al personal de seguridad, grabar, etc.)[12].

1.2 Motivación

A pesar de no reducir la delincuencia en gran medida, es verdad que la videovigilancia es esencial para la seguridad, dado que más que reducir la violencia, la previene, pues la colocación de cámaras y el aviso de ésta provocan un efecto disuasorio y, en caso de que se cometa el delito, representan una prueba definitiva ante el juez, y mucho mejor si las cámaras disponen de analítica de vídeo, pues ahorrarán bastante tiempo al personal de seguridad debido a que no se requiere revisar horas y horas de vídeo.
Ahora, el usuario que quiera colocar cámaras se enfrenta a otro problema, y es que éste no puede ver la eficacia de las cámaras a simple vista, pues cada cámara tendrá un campo de visión diferente según sus características (resolución, distancia focal, tamaño del objeto a detectar, etc.) y corre el riesgo de colocar una cantidad de cámaras innecesaria, aumentando así los gastos para mantener la seguridad de su hogar, negocio, etc. Por esto, el sistema a elaborar se encargará de proporcionar una serie de cámaras con características diferentes que, al colocarse sobre un determinado recinto representado en un mapa, mostrarán un campo de visión determinado dependiendo de sus características y del objeto que se quiera detectar. De esta forma, cualquier persona que quiera colocar cámaras alrededor de su negocio, hogar, etc., utilizará la cantidad justa de cámaras para garantizar la mayor seguridad posible gastando lo mínimo necesario. Dado el gran aumento del mercado en cámaras de seguridad (especialmente de las que disponen de analítica de vídeo), esta aplicación puede ser no sólo útil, sino esencial si se desea vigilar de manera eficaz cualquier tipo de recinto.

1.3 Objetivos

Con esta aplicación, se busca cumplir con los siguientes objetivos:

- **Capacidad de registrar nuevos modelos de cámaras con características diferentes**: Habrá un usuario administrador que cree nuevos modelos de cámara, para poder tener varias opciones a la hora de colocar las cámaras sobre el mapa.
- **Capacidad de modificar o eliminar modelos de cámaras existentes en el sistema**: Con esto se quiere dar la opción al usuario de modificar las características de una cámara en caso de que no haya quedado satisfecho con las mismas, o incluso eliminar una o varias cámaras ya sea por borrar cámaras innecesarias, petición del cliente, etc.
- **Disponer de varios perfiles de usuario**: Para el caso de este sistema, se dispondrán de dos perfiles, el del administrador y el del usuario común. Esto es importante, pues garantiza una buena organización de funcionalidades, y de que sólo gente cualificada en cámaras de seguridad sea la que cree nuevos modelos de cámara en la aplicación (es por esto que sólo hay UN usuario administrador).
- **Representar la información de forma gráfica**: Mostrar la lógica del sistema de forma gráfica de manera que sea agradable y fácil de utilizar para el usuario.
- **Disponer de herramientas de colocación intuitivas sobre el mapa**: El usuario debe saber cómo colocar una cámara sobre el mapa, pues de primeras éste no sabrá si hay que hacer click sobre una cámara determinada y luego pulsar sobre el mapa, arrastrar la cámara hasta el mapa, etc.
- **Poder exportar la información para entregar al cliente**: El sistema debe tener la funcionalidad de exportar el proyecto en una o varias imágenes para que el cliente pueda tener guardado un documento visual que muestre de forma realista la eficacia que tendrían las cámaras que haya elegido para proteger su recinto.
- **Mantenimiento sencillo**: La aplicación deberá ser modular y tener código bien organizado para facilitar su mantenimiento y el desarrollo de futuras versiones de la misma.

Además, se busca el aprendizaje de nuevas tecnologías como Qt, conocer las ventajas que ésta tiene frente a otros entornos de trabajo para la elaboración de aplicaciones con interfaz gráfica de usuario, aprender a utilizar otros lenguajes de programación como C++, QML con Javascript, etc.

1.4 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Tecnologías a utilizar:** Aquí se hablará sobre el entorno de programación que se ha utilizado (junto con otras tecnologías).
- **Diseño:** Aquí se profundizará en el diseño y estructura que se ha seguido en la implementación de la aplicación.
- **Desarrollo:** En este apartado se profundizará en el desarrollo, en donde se incluirán los pasos seguidos en el proyecto, se mostrará la navegación de la aplicación, y se discutirán todos los problemas encontrados y cómo han sido solucionados.
- **Pruebas y resultados:** En esta sección se discutirán las pruebas realizadas sobre el sistema para la detección de posibles errores y los resultados que éstas han tenido.

- **Conclusiones:** En este apartado se hablará finalmente de lo aprendido con el proyecto, si se han cumplido los objetivos iniciales del mismo y la utilidad que ha tenido para su autor.
- **Referencias:** Aquí se incluirá una serie de fuentes en las que el autor se ha basado para el desarrollo del sistema.

2 Tecnologías a utilizar

Para este proyecto, se ha hecho uso de varias tecnologías, las cuales se explicarán a continuación:

- **Qt:** En el verano de 1990, Haavard Nord y Eirik Chambe-Eng estaban trabajando juntos en una aplicación de base de datos en C++ para imágenes de ultrasonido, el cual debía ser multiplataforma y ejecutarse con una interfaz gráfica de usuario. Un día, sentados en un banco, Haavard dijo que necesitaban un sistema gráfico orientado a objetos… ese fue el nacimiento intelectual de Qt. Más tarde, el 20 de mayo de 1995, se llevaría a cabo el primer lanzamiento público de Qt[30].

  [10]Qt es un entorno de trabajo multiplataforma orientado a objetos. Es bastante utilizado para desarrollar programas con interfaz de usuario, así como también diferentes tipos de herramientas para la línea de comandos y consolas para servidores que no requieren interfaz de usuario. Utiliza principalmente el lenguaje de programación C++, aunque también puede ser usado en varios lenguajes de programación (a través de **bindings**). La API de su biblioteca permite que acceder a bases de datos SQL, utilizar XML, gestión de hilos, soporte de red, etc.

  Se optó por usar esta tecnología frente a otras por diversos motivos:
  - Tiene una gran cantidad de documentación que ha sido acumulada por 25 años, además de contar con una basta comunidad activa de millones de usuarios dispuestos a ayudar con cualquier duda y/o problema.
  - Como se ha mencionado, es un entorno multiplataforma, lo que permite que las aplicaciones que se desarrollen utilizando Qt funcionen en diversos sistemas operativos. Por este motivo, grandes compañías como la Agencia Espacial Europea, Volvo Mobility Systems, Panasonic, etc., usan este *framework*.
  - Uno de los motivos más importantes que hicieron Qt preferente frente a otras tecnologías para este proyecto es que dispone de un editor visual para desarrollar la interfaz gráfica de usuario de manera cómoda y sencilla; así cosas como el ajuste de las diferentes características de los elementos de la interfaz se realizan de manera bastante más rápida. Esto hace que este *framework* sea ampliamente utilizado y preferible para el desarrollo de aplicaciones con interfaz gráfica de usuario.
  - También ofrece Qt Quick, un kit que contiene un lenguaje llamado QML, el cual permite el uso de JavaScript para proporcionar la lógica a la misma interfaz de usuario, además de crear otros elementos GUI.

- **VirtualBox:** Software de virtualización para arquitecturas x86/amd64. Actualmente es desarrollado por Oracle Corporation como parte de su familia de productos de virtualización (motivo por el que su título oficial es Oracle VM VirtualBox), pero en sus orígenes fue desarrollado por Innotek GmbH (empresa alemana), luego fue vendido a Sun Microsystems[3].

En 2010, VirtualBox fue el software de virtualización más utilizado.
Incluye una gran cantidad de sistemas operativos, como Windows XP, Windows Vista, Windows 7, macOS X, Linux, Solaris, etc. Esta herramienta se ha utilizado durante el desarrollo del proyecto para instalar y ejecutar el sistema operativo Linux, en el que se ha hecho la aplicación (se ha optado por desarrollar la aplicación en este sistema operativo dado la familiarización que se tiene con el mismo para la ejecución de diversos comandos que se encargaron de la compilación y ejecución de la aplicación, así como el uso de comandos para la instalación de varias dependencias que requería el *framework* Qt).

Uno de los motivos principales por los que se ha utilizado es que permite instalar de forma rápida y sencilla diversos sistemas operativos sin necesidad de realizar particiones del disco duro de la máquina anfitrión. Durante el proyecto hubo un fallo en la partición en la que solía estar Ubuntu, y hubo fallos posteriores cuando se intentó volver a instalarlo en una partición de disco duro junto a Windows 10, por lo que se optó por VirtualBox como solución fácil y rápida. Además, este software no solo permitía una transición rápida entre sistemas operativos sin necesidad de apagar el ordenador, sino que también permitía arrastrar y soltar ficheros y copiar diversos elementos de forma bidireccional. Es decir, se podía arrastrar y soltar ficheros de la máquina anfitrión a la máquina virtual y viceversa; lo mismo para copiar diferentes elementos como texto, ficheros, etc.

- **SQLite Studio:** Interfaz que permite gestionar bases de datos SQLite. SQLite es un sistema de gestión de bases de datos relacional compatible con ACID, esto quiere decir que las transacciones en SQLite son tratadas como una unidad, que éstas vayan de un estado válido a otro, la ejecución concurrente se realice de forma correcta como si de una ejecución secuencial se tratase y que las transacciones completadas sean guardadas en una memoria no volátil. Este gestor está contenido en una pequeña biblioteca programada en C[11].
  Se ha optado por el gestor SQLite dada la poca cantidad de memoria que ocupa, además es más rápido que otros gestores de bases de datos como MySQL y PostgreSQL, y es multiplataforma, por lo que sus bases de datos se pueden portar a otros sistemas operativos sin necesidad de ninguna configuración previa.

A parte de haber utilizado estas tecnologías, la aplicación se ha escrito en los siguientes lenguajes de programación:

- **C++:** Diseñado en 1979 por Bjarne Stroustrup. Extiende al lenguaje C en el sentido de que C++ permite la manipulación de objetos, por lo que se considera un lenguaje híbrido desde el punto de vista de los lenguajes orientados a objetos.
- **QML:** (acrónimo de Qt Meta Language) Lenguaje basado en JavaScript creado para diseñar aplicaciones enfocadas a la interfaz de usuario. Forma parte de Qt Quick, que es un kit de interfaz de usuario creado junto con el entorno de desarrollo Qt.

3 Diseño

Ahora se va a explicar de forma detallada el diseño y desarrollo del proyecto. Se han utilizado los siguientes módulos y ficheros para el desarrollo de la aplicación:

3.1 Módulos (ficheros .cpp y .h):

3.1.1 Módulo Main

Este es el módulo principal que ejecuta la aplicación en su totalidad.

3.1.2 Módulo Mainwindow

Este módulo se encarga de gestionar la ventana principal y única de la aplicación, además de la lógica general de la misma. Este módulo es instanciado en el fichero "main.cpp" para luego mostrar la ventana.

3.1.3 Módulo Dragwidget

Este módulo extiende de QFrame y es el encargado de gestionar el contenedor en donde se van a encontrar las cámaras que el usuario común puede seleccionar y colocar sobre el mapa.

3.1.4 Módulo Camera

Este módulo únicamente contiene toda la información de cada cámara, como su nombre, su distancia focal, la anchura del sensor (en metros), la altura del sensor (en metros), la anchura de la resolución (en píxeles) y la altura de la resolución (en píxeles).

3.2 Otros ficheros

3.2.1 Mainwindow.ui

Fichero que contiene la mayoría de los elementos gráficos que se usan en la aplicación (otros son añadidos en mainwindow.cpp). El elemento GUI más importante aquí es un widget llamado Stacked Widget, el cual se explicará más adelante.

3.2.2 Mapviwer.qml

Fichero que gestiona toda la lógica y representación gráfica del mapa.

3.3 Organización de las clases

Para analizar de una forma simple el contenido de estas clases y cómo interactúan entre sí, se presenta el siguiente diagrama de clases.

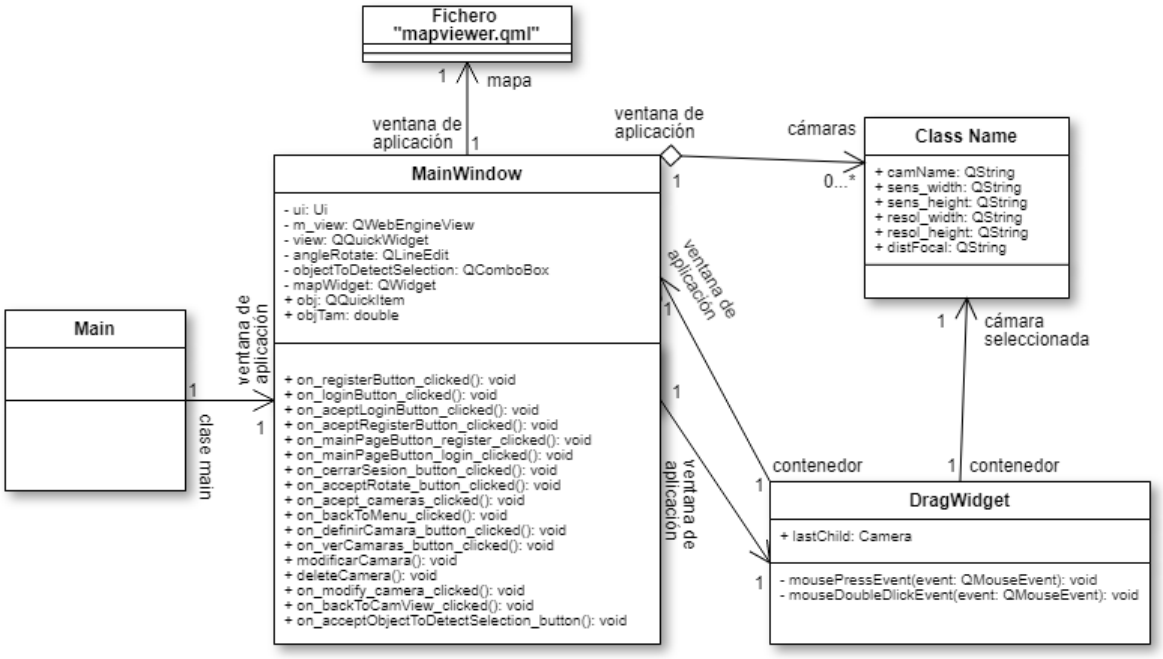

Figura 1. Diagrama de clases de la aplicación SafeWatch

Por cuestiones de simplicidad, se ha descartado mostrar algunos métodos cuya lógica era exactamente igual, pero son utilizados en diferentes contextos, por lo que sus nombres difieren.

3.4 Base de datos

El sistema utiliza una base de datos relacional SQLite que contiene dos tablas:

3.4.1 Users

Tabla que contiene la información del usuario en 3 columnas: un nombre, una contraseña y un valor booleano que indica si el usuario es administrador o común.

3.4.2 Cameras

Tabla que contiene la información de la cámara en 6 columnas: un nombre, una distancia focal, una anchura de sensor (en metros), una altura de sensor (en metros), una anchura de resolución (en píxeles) y una altura de resolución (en píxeles).

Esta base de datos se puede representar gráficamente con el siguiente diagrama:

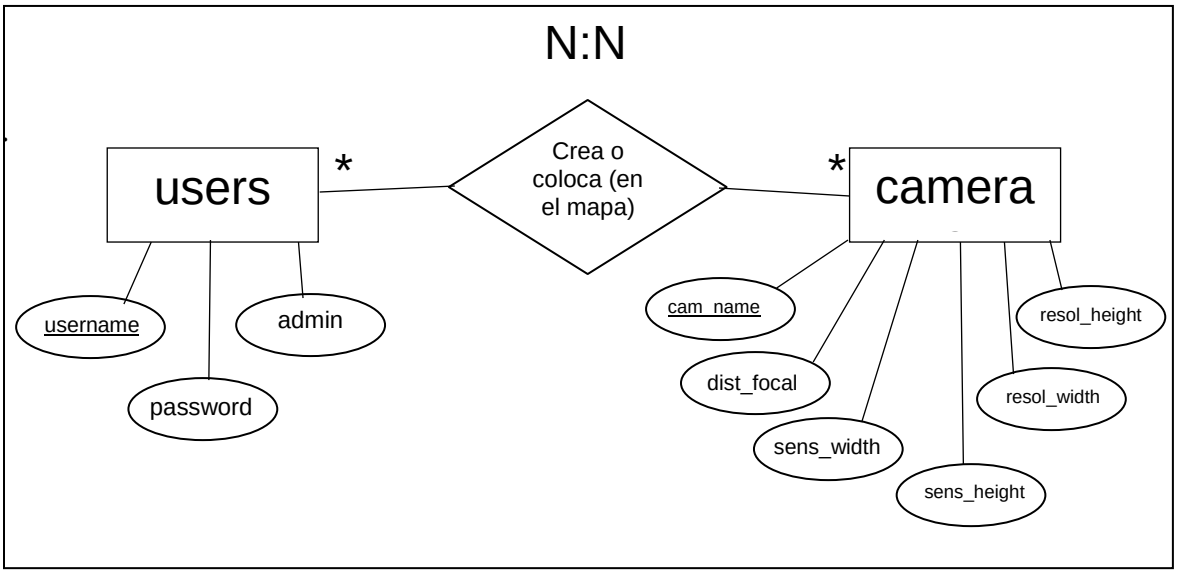

Figura 2. Diagrama del modelo entidad-relación de la base de datos del sistema

3.5 Casos de uso

Como ya se ha explicado anteriormente, la aplicación va a disponer de dos perfiles de usuario: el administrador y el usuario común. A continuación, se van a mostrar las funcionalidades que tiene cada uno dentro del sistema a través del siguiente diagrama de casos de uso.

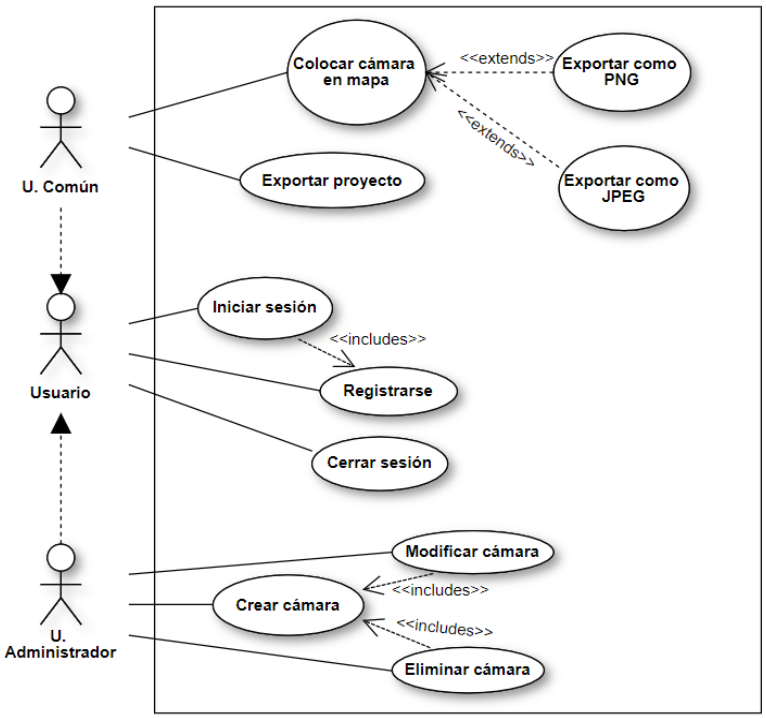

Figura 3. Diagrama de casos de uso (todos los casos de uso, excepto "Registrarse", tiene una dependencia de tipo "includes" hacia "Iniciar Sesión")

3.6 Estructura las vistas

Ahora se pasará a explicar de forma detallada la estructura general del proyecto y el cómo se han organizado las vistas o widgets que representan las diferentes "páginas" de la aplicación.
Lo primero ha sido organizar el diseño del sistema; una de las preguntas más importantes que se plantearon acerca de esta cuestión fue el cómo organizar la interfaz de manera que se pueda pasar de un widget a otro de manera sencilla. Finalmente, se optó por utilizar el Stacked Widget[29], el cual es un widget que organiza varios widgets de manera muy similar a como hace el CardLayout de Java, es decir, sólo se puede ver uno a la vez, y es fácil indicar cuál mostrar (basta con llamar al método "setCurrentIndex" de la instancia de Stacked Widget).
A continuación, se muestra de manera resumida cómo está organizado el Stacked Widget que se utilizó para la aplicación, el cual se llama "stackedWidget":

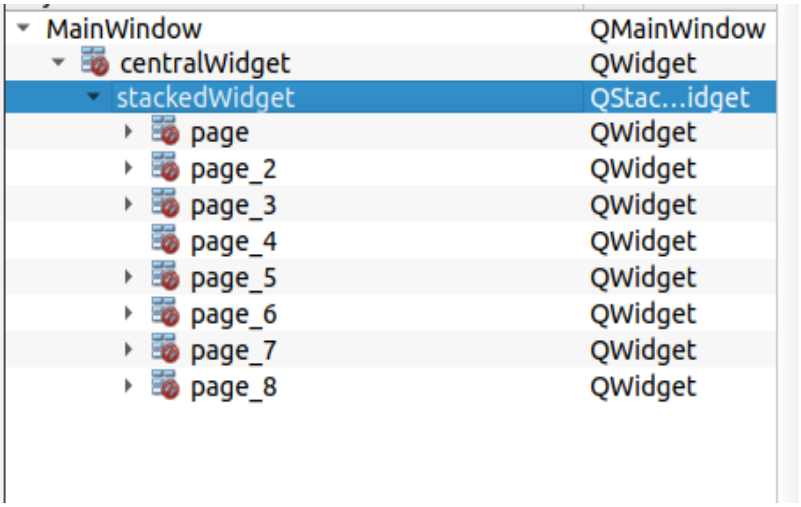

Figura 4. Estructura del Stacked Widget mostrada en la ventana del diseñador de Qt.

La mayoría de los widgets (definidos como "page", "page_2", etc.) fueron diseñados en el propio diseñador que proporciona Qt (en este caso, **mainwindow.ui**). Otros, al contener elementos que cambiarían forma dinámica, fueron diseñados a través de código en el fichero **mainwindow.cpp**. Ahora se pasará a describir de forma reducida qué representan cada uno de los widgets:

- **Page**: Menú principal de la aplicación, con las opciones de "Registrarse" e "Iniciar sesión"
- **Page_2**: Página que muestra el formulario para el registro de nuevos usuarios.
- **Page_3**: Página que muestra el formulario para el inicio de sesión de usuarios ya dados de alta en el sistema.
- **Page_4**: Interfaz del usuario común, en la que puede colocar diferentes modelos de cámaras ya existentes sobre un mapa, así como girarlas, y seleccionar el objeto de éstas deben detectar.
- **Page_5**: Menú principal del usuario administrador, con las opciones de ver las cámaras que ha creado y definir un modelo de cámara nuevo.

- **Page_6**: Interfaz del administrador en donde puede ver la información de las cámaras que ha creado. También dispone de las funcionalidades de modificar y eliminar la/s cámara/s que éste desee.
- **Page_7**: Página que muestra el formulario para definir una cámara nueva en el sistema.
- **Page_8**: Página que muestra el formulario para modificar una cámara determinada. Es un formulario idéntico al del widget antes mencionado.

3.7 Fórmulas (óptica)

Una de las mayores incógnitas para el desarrollo del sistema es la visualización gráfica del campo de visión de las cámaras que se coloquen sobre el mapa. Al crear una nueva cámara en la aplicación, se definen la distancia focal, el tamaño del sensor y la resolución, por lo que estos datos son los que se conocen desde el principio. Con estos datos y los debidos conocimientos en óptica, se pueden sacar un par de fórmulas que serán esenciales a la hora de calcular el campo de visión de la cámara.

La primera fórmula es útil para calcular la distancia máxima que puede detectar la cámara[16].

$$d = \frac{T_o * d'}{T_i}$$

T_o es el tamaño del objeto a detectar, d' la óptica o distancia focal y T_i es el tamaño de la imagen, cuya fórmula es la siguiente:

$$Ti = num_{pixeles}\ x\ tam_{pixel}\ x\ 5$$

tam_{pixel} corresponde al tamaño del píxel (que es igual a la anchura del sensor partido por la anchura de resolución), $num_{pixeles}$ es el número de píxeles, los cuales, para tener un mínimo de tamaño de imagen a detectar en el sensor, serán 5.

La siguiente fórmula será útil para obtener el ángulo de apertura horizontal de la cámara[33]:

$$2 * \text{atan}(\frac{sens_{anch}}{2 * d'})$$

En esta fórmula, $sens_{anch}$ vendría a ser la anchura del sensor en metros y d' la distancia focal. Esto finalmente nos devolverá el ángulo de apertura horizontal de la cámara en grados.

4 Desarrollo

A continuación, se pasará a explicar de manera detallada el procedimiento seguido para llevar a cabo cada una de las funcionalidades principales de la aplicación.

4.1 Implementación del login

Primero, como se mencionó antes, se decidió almacenar a los usuarios dados de alta en la aplicación en una base de datos SQLite. Cuando se registran, se insertan en ésta, y cuando se "loguean" al sistema, se consulta a la base de datos si existe el usuario, y si la consulta devuelve un resultado, el inicio de sesión se da por válido.
Por simplicidad y diseño, se decidió que sólo va a haber un usuario administrador previamente insertado en la base de datos, con un valor de **true** en la columna "admin" (de tipo booleano); por defecto, el sistema de registro que proporciona la aplicación inserta al usuario en la base de datos con valor **false** en la columna "admin". De este modo, el sistema de login comprueba el valor de la columna "admin" del usuario; si ésta es **true**, la aplicación carga la interfaz del administrador, y en caso contrario, carga la interfaz del usuario común.

4.2 Implementación del perfil del administrador

4.2.1 Menú principal

En la interfaz del administrador, empezamos con un menú que tiene dos botones: uno que lleva a una vista que muestra todas las cámaras creadas con la información detallada de cada una, y otro que lleva a otra interfaz para crear otra cámara.

4.2.2 Formulario para crear un modelo nuevo de cámara

Si se pulsa en la segunda opción, se mostrará una vista con un formulario en el que se introducen todos los datos que se desea que tenga la cámara. Una vez se pulsa en "Aceptar", la cámara se inserta en la base de datos habiendo realizado las conversiones necesarias, pues el tamaño del sensor se selecciona en pulgadas, pero interesan las medidas de anchura y altura en metros, y de la resolución interesan la anchura y la altura en píxeles. De esta forma, la cámara que haya sido creada tendrá la posibilidad de poder ser leída de la propia base de datos por cualquier widget de la aplicación.

4.2.3 Implementación del menú de cámaras existentes

Volviendo al menú principal del usuario administrador, si se pulsa en la primera opción, se puede ver un listado en donde se pueden apreciar de forma ordenada las cámaras creadas, las cuales se han organizado, dentro de un contenedor de tipo Scroll Area[28], como un Vertical Box Layout, y cada fila de ese layout consta de 3 elementos (organizados a su vez en un Horizontal Box Layout): el nombre de la cámara, información de la misma, y dos botones que permiten modificar los datos de la cámara o borrar la misma respectivamente. Esta vista con cámaras creadas se ve de la siguiente manera:

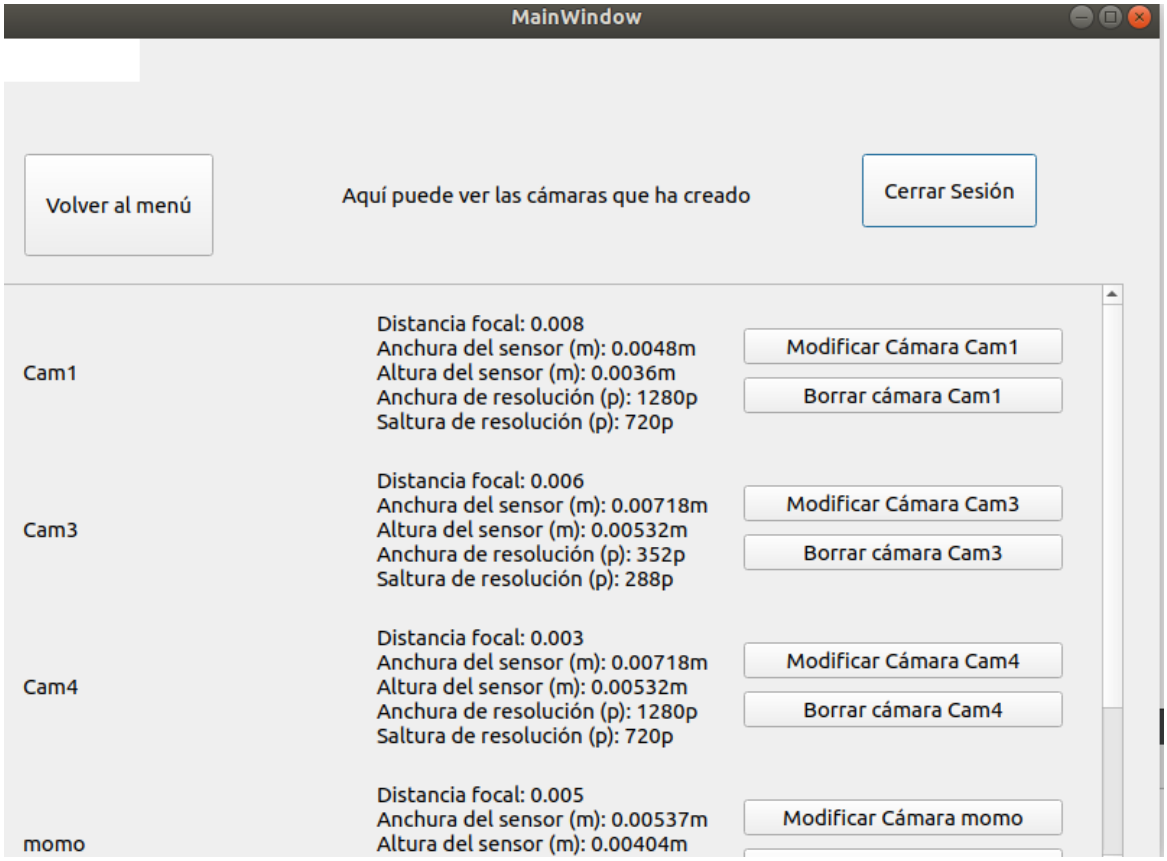

Figura 5. Vista del widget "page_6" con cámaras creadas.

Se realiza una consulta general de todas las cámaras que estén en la base de datos, y así, se hace un bucle **while** tantas veces como filas devueltas por la consulta haya, y en cada vuelta, al layout horizontal de cada fila se añade el nombre de la cámara, la información de ésta y los botones, y este layout se asigna a un widget que a su vez será añadido a un layout vertical, el cual es asignado al widget contenido dentro del "Scroll Area".
Los botones por su parte sirven para modificar o eliminar la cámara que se encuentra en la misma fila.

4.2.3.1 Implementación para modificar la cámara

Si se pulsa en “Modificar cámara…”, lleva a una vista casi idéntica a la que crea una cámara, sólo cambia en un par de etiquetas y en que, en vez de insertar la cámara en la base de datos, se actualiza la cámara correspondiente en la base de datos con la nueva información.

4.2.3.2 Implementación para eliminar la cámara

Por el otro lado, el botón de “Borrar cámara…” únicamente muestra un diálogo preguntando al usuario si está seguro de eliminar la cámara; si acepta, la cámara es eliminada de la base de datos (mediante una query DELETE) y, por ende, de la aplicación.

4.3 Implementación del perfil del usuario común

Ahora se pasará a describir el desarrollo de la interfaz del usuario común, que, aunque sólo consista en una sola página, es la más importante y la que más trabajo ha requerido de todo el proyecto. Esta vista luce de la siguiente manera:

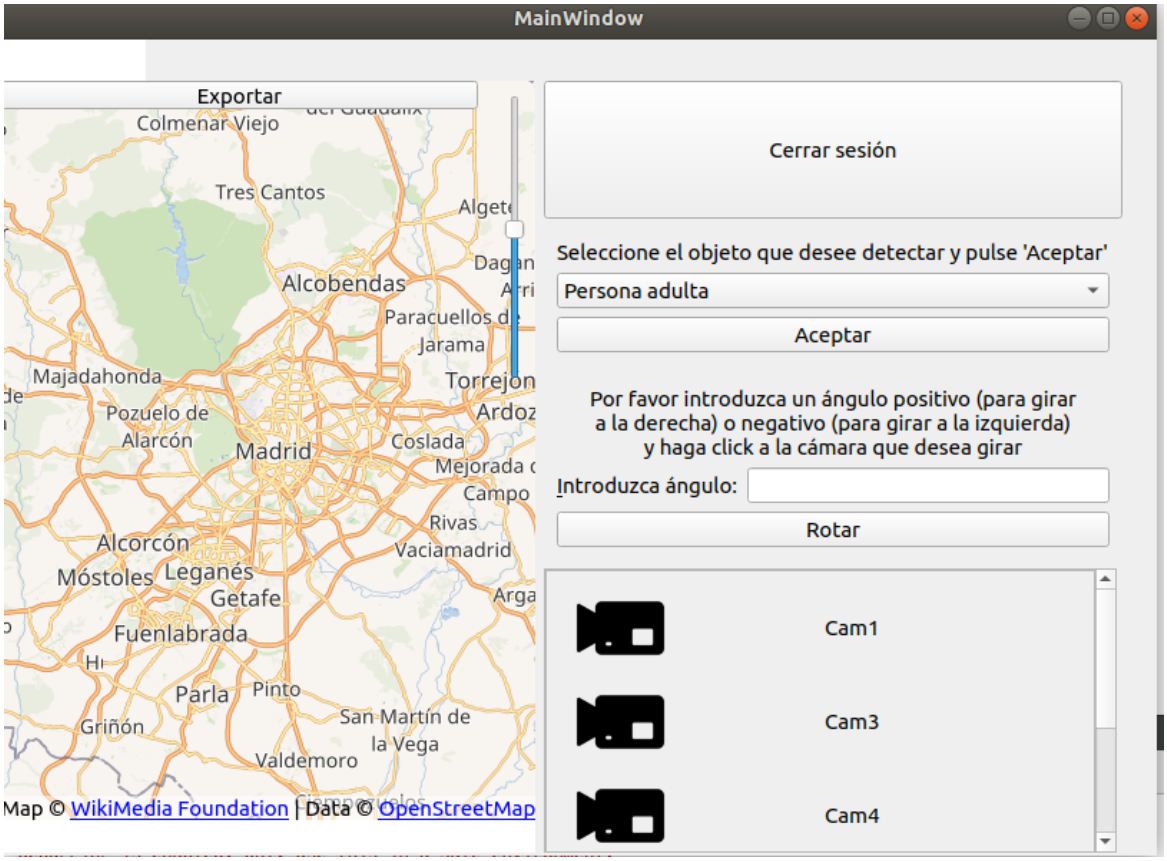

Figura 6. Vista de “page_4”, donde el usuario puede hacer sus proyectos de videovigilancia.

4.3.1 Implementación del mapa

El primer paso fue cargar el mapa dentro de la ventana. Para ello, se instanció un widget de tipo Quick (QQuickWidget[27]), el cual se llama **view**; es de tipo Quick debido a que esta clase, que hereda de la misma QWidget, tiene los métodos necesarios para enlazar el fichero QML (en este caso será **mapviewer.qml**, el cual se encarga de mostrar gráficamente todo lo que ocurre en el mapa, así como su lógica) con el propio widget **view**. Para desarrollar el mapa se obtuvo código de los ejemplos ofrecidos por la herramienta Qt Creator, concretamente, el ejemplo de **Map Viewer**[21].

4.3.2 Implementación del contenedor de cámaras existentes

Después, se creó el contenedor que mantendría las cámaras creadas, el cual, de forma similar que con el mapa, se pudo desarrollar con la ayuda del código del proyecto de ejemplo **Draggable Icons**[14] proporcionado por Qt Creator. Este widget ha sufrido varias modificaciones a lo largo del proyecto, las cuales se explicarán en detalle más adelante cuando se hable de los problemas encontrados durante el proyecto y cómo han sido solucionados. Este es otro widget que, junto con el que mostraba las cámaras en la interfaz del administrador, lee todas las cámaras de la base de datos y las muestra, con la peculiaridad de que enseña únicamente la imagen de la cámara a la izquierda y su nombre a la derecha, pero se puede acceder a su respectiva información pulsando dos veces sobre la imagen.

4.3.3 Primera implementación para colocar cámaras en el mapa

Habiendo puesto el contenedor de cámaras, el paso siguiente fue conseguir que se colocasen las cámaras en el mapa, lo cual se hace fácilmente al pulsar sobre el propio mapa, y la cámara se quedará anclada en el mismo con las coordenadas de donde se pulsó el ratón. Para que se pudiesen añadir más cámaras, se utilizó un modelo de lista, de esta forma se pueden añadir varias cámaras que se mostrarán en el mapa y cada una es independiente del resto.

4.3.4 Primera implementación para mostrar el campo visual de la cámara

Una de las cosas más complicadas dentro de esta página ha sido representar el campo visual de la cámara. Para éste, se han utilizado polígonos de mapa (MapPolygons en QML[19]; son elementos que proporcionan los elementos de tipo “Map”). Para probar, se empezó colocando 3 puntos de manera fija (con uno de ellos en el centro de la cámara). Lo siguiente fue conseguir unir el campo visual con su respectiva cámara de cara a que, al arrastrar la cámara por el mapa, también se arrastrase su respectivo campo visual. La mayoría de los aspectos que se desarrollaron aquí se explicarán en detalle más adelante en “Problemas encontrados y soluciones”.

4.3.5 Implementación de la rotación de las cámaras en el mapa

Otro aspecto importante que se tuvo en cuenta fue el girar las cámaras. Esto se detallará mejor en el subapartado "Problemas encontrados y soluciones", pero finalmente se decidió hacerlo a través de un formulario, en el que se inserta una cantidad en grados, y se da click a un botón para enviar esa información al mapa, información que se usará para rotar una cámara (junto con su campo visual) la cantidad de grados que se introdujo en el formulario.

4.3.6 Implementación del paso de información al fichero QML

El paso siguiente fue pasar la información de una determinada cámara (la cual se pasa seleccionándola) al fichero QML del mapa. Esto ha sido posible gracias al método "setProperty". Para esto, primero se creó una instancia de QQuickItem[26] (Esta instancia es estática… en el subapartado "Problemas encontrados y soluciones" se explicará el porqué de esta decisión), se le asigna el objeto raíz del widget **view** (recordemos que este widget estaba enlazado con el fichero QML del mapa), y con esta instancia ya podemos llamar al método antes mencionado, pasándole dos parámetros: el nombre de la propiedad del fichero QML a la que queremos asignarle un valor y el valor que queremos asignar a la misma propiedad. (Esto mismo es lo que se hace para pasar la información de los grados que se desea girar la cámara). Aquí, cuando se selecciona una cámara, se pasa la información de la distancia focal, la anchura y altura del sensor y la altura de la resolución.

4.3.7 Implementación final para mostrar el campo visual de la cámara

Para resolver este problema, ya se dio un gran paso en el apartado de diseño cuando se elaboraron unas pocas fórmulas para poder calcular la distancia máxima de la cámara y su ángulo de visión… ahora se deben implementar. Todos los cálculos para calcular el campo de visión de la cámara se han realizado en **mapviewer.qml**.

Al hacer click sobre la cámara, ya se envían a **mapviewer.qml** los datos esenciales como la distancia focal, el tamaño del sensor, la distancia máxima y la anchura de la resolución, pero aún falta por saber el tamaño del objeto que se quiere detectar.

Para ello, arriba del formulario para introducir el ángulo para girar la cámara, se encuentra una pequeña sección en donde se selecciona el objeto que se desea detectar (los objetos de la lista ya tienen una altura asignada), y finalmente, cuando se pulsa en el botón de "Aceptar", se envía la información de la altura del objeto a detectar de la misma manera que se ha explicado anteriormente con el método "setProperty" llamado por la instancia de QQuickItem.

Una vez que se hayan aplicado las fórmulas de óptica necesarias, queda mostrar el campo de visión final de la cámara que se coloca sobre el mapa. Para visualizar este campo de visión se utiliza un polígono en el mapa, el cual tiene 3 puntos de coordenadas. En el instante en que se coloca la cámara en el mapa, el polígono tiene sólo un punto correcto (el centro de la cámara, o lo que es lo mismo, donde se pulsa sobre el mapa); a éste se le llamará "A", los otros dos de momento se considera que están en la misma línea horizontal (misma latitud, pero diferente longitud); a éstos se les llamará "B" y "C". Nos falta otro dato útil, y es la distancia que habría desde el punto "A" a cualquiera de los otros dos. Trigonométricamente, se puede hallar de la siguiente forma:

$$distToPoint = \frac{d}{\cos \alpha}$$

(Se considera “d” (distancia máxima a la que detecta la cámara) como si fuese la altura de un triángulo isósceles y “α” como la mitad del ángulo de visión).
Ahora se tienen los datos necesarios y conocimientos en trigonometría: uno de los puntos (que es en donde se coloca la cámara), la distancia desde “A” a cualquiera de los otros dos puntos y la mitad del ángulo de apertura horizontal de la cámara.

[1]Para calcular los otros dos puntos del polígono, vamos a necesitar de un ángulo al que llamaremos **bearing**. El bearing se calcula con el arco tangente de dos dimensiones entre un valor X y un valor Y, los cuales se calculan de la siguiente forma:

$$X = \cos \theta * \sin \Delta L$$

$$Y = \cos \theta a * \sin \theta b - \sin \theta a * \cos \theta b * \cos \Delta L$$

Se consideran “L” como la longitud, “**θ**” como la latitud, “a” como el punto inicial y “b” como el punto final. En este caso, se supone que se busca primero el bearing entre el punto “A” y el punto “B”, y después se busca el bearing entre el punto “A” y el punto “C”. Ahora ya se conocen el punto de partida “A”, los dos bearings y la distancia desde el punto inicial a cualquiera de los otros dos (“B” y “C”). Ahora, para calcular las nuevas coordenadas de los puntos “B” y “C”, se utiliza un método auxiliar llamado “calculateCoord”[2], el cual, a partir de las coordenadas del punto inicial, el bearing (en este caso, vamos a pasarle al método la suma o resta, dependiendo del punto del triángulo que se quiera calcular, del bearing con la mitad del ángulo de apertura de la cámara) y la distancia a ese punto, se calculan las coordenadas de éste último. Al aplicarse este método auxiliar a ambos puntos “B” y “C”, obtenemos sus nuevas coordenadas, y de esta forma ya se consigue una representación visual más exacta y realista del campo visual de la cámara. Cabe destacar que este método se utilizó primero cuando se resolvió el problema de girar las cámaras junto con tu campo visual… esto se explicará más adelante en “Problemas encontrados y soluciones”.

4.4 Problemas encontrados y soluciones

1. Problema #1:

   En un primer momento, el mapa no se mostraba en la zona que debería estar.

   Solución:

   Se hizo uso del proyecto de ejemplo que proporciona Qt Creator llamado “Map Viewer” para mostrar un mapa, aunque en el proyecto de ejemplo, el elemento raíz del fichero QML donde se hallaba el mapa era de tipo AppWindow (ventana de aplicación), por lo que se cambió eso por “Item”. A partir de ahí, el mapa se mostró de forma correcta.

2. Problema #2:

   Al principio, se planteó colocar las cámaras en el mapa a través de la acción de "arrastrar y soltar".

   Solución:

   De forma similar a la solución del problema anterior, se utilizó otro proyecto de ejemplo de Qt, en este caso, "Draggable Icons", el cual tiene un contenedor con varias etiquetas arrastrables y soltables, así que se usó el módulo que instancia este contenedor para aplicarlo a este proyecto. Al principio, este contenedor contenía por defecto una etiqueta en forma de la imagen de una cámara, la cual se podía arrastrar y soltar. Por su parte, en el fichero **mapviewer.qml** se gestionaba la lógica de soltar la cámara, de manera que, al soltar un objeto en el mapa, éste creaba un ítem (el cual tenía la imagen de una cámara) en el punto donde se soltaba la cámara.

3. Problema #3:

   Después de solucionar el problema de fijar una cámara en el mapa, se planteó cómo fijar varias cámaras en el mapa de manera que cada una fuese independiente de las demás.

   Solución:

   Al principio se crearon métodos auxiliares en Javascrpt que creasen otros ítems de mapa con la imagen de una cámara. Pero al final se optó por una solución más sencilla: crear un modelo de lista para las cámaras (llamado **mapModel**)[18]. De esta forma, cada cámara era un "MapQuickItem"[20] dentro de un "MapItemView", y cada vez que se colocaba una cámara en el mapa, se añadía al modelo antes mencionado pasándole como argumento la latitud y longitud en donde se colocaba la cámara. Así, en el mapa se veían todos los elementos del **mapModel**.

4. Problema #4:

   Al intentar colocar un campo visual (por defecto, de momento) a la cámara, el elemento "MapQuickItem" no permitía añadir otro elemento (pues ya estaba el elemento que representaba la imagen de la cámara). ¿Cómo añadir el campo visual junto con la cámara?

   Solución:

   De forma casi idéntica que con la solución anterior, se creó un modelo de lista para los campos visuales (llamado **cameraViewModel**), para que se añadiesen a éste al mismo tiempo que se colocaba la cámara en el mapa, pasándole como argumentos la latitud y longitud de donde se ponía la cámara.

5. Problema #5:

   En relación con la solución anterior, al mover la cámara, no se movía su respectivo campo de visión, pues las cámaras y los campos visuales estaban en dos listas diferentes.

   Solución:

   Todo dependía de la propiedad "drag.target" que tiene el "MouseArea"[22] que se localiza dentro del "MapQuickItem" que representa la cámara, por lo que había que plantearse cómo hacer para que el drag.target sea múltiple. Finalmente se optó por crear un modelo de lista que contuviese ítems del tipo "MapQuickItem" (llamado **dragModel**). De esta forma, cuando se añadiesen al mapa tanto la cámara como su campo visual, también se añadiría este elemento nuevo, cuya función consiste básicamente en contener a los dos elementos antes mencionados y, cada vez que cambia la posición de este último elemento, se modifica la posición de los dos anteriores.

6. Problema #6:

   Por motivos de implementación, para rotar la cámara finalmente se optó por introducir mediante un formulario la cantidad de grados en que se desearía girar la cámara con cada click, pero antes de averiguar cómo pasar la información del fichero **mainwindow.cpp** al fichero **mapviewer.qml**, se cuestionó cómo girar la cámara junto con su campo visual, pues, a pesar de que girar la cámara es bastante sencillo gracias a la propiedad "transform" de "MapQuickItem", girar el campo visual de la cámara fue bastante complicado. Finalmente, se concluyó de que había que cambiar directamente los puntos del polígono.

   Solución:

   Se conocen varios datos, como el bearing inicial entre el punto inicial "A" (el centro de la cámara) y el otro punto (cualquiera de los otros dos, "B" o "C"), la distancia entre estos dos y el ángulo que se desea girar. Acto seguido, se hizo uso del método auxiliar en Javascript ya explicado llamado "calculateCoord", al que se le pasan los datos antes mencionados y devuelve la latitud y la longitud del nuevo punto " B' " o " C' " (según cuál se "gire"). Este método se llama un total de 4 veces, 2 para la latitud y longitud del punto " B' " respectivamente y otras 2 para la latitud y longitud del punto "C' " respectivamente. Finalmente, se modifica el polígono con las nuevas coordenadas.

7. Problema #7:

   Justo después de resolver el problema anterior, vino el problema de cómo pasar, del fichero **mainwindow.cpp** al fichero **mapviewer.qml**, la información de la cantidad de grados a girar una cámara determinada.

Solución:

La clase “QObject” tiene un método llamado “setProperty”, al cual se le pasa el nombre de la propiedad que esté en el fichero QML, y el valor a asignar a esa propiedad. Primero, se ha hecho una instancia de dicha clase (llamada **obj**), y luego se le asignó el objeto raíz de la instancia **view** (es la que tiene asignada la fuente del fichero **mapviewer.qml**); en otras palabras, a **obj** se le asignó el elemento raíz de **mapviewer.qml**, que en este caso es el elemento QML de tipo “Item”. Una vez hecho esto, ya se puede llamar al método “setProperty” pasándole los argumentos adecuados.

8. Problema #8:

   Habiendo resuelto todos los problemas anteriores, sólo faltaba pasar la información de la cámara al fichero **mapviewer.qml** para que, según la cámara seleccionada y el tamaño del objeto que se quiera detectar, se muestre una cámara con su campo visual en función de las características de la misma. Dada la implementación seguida hasta ahora, realizar esta acción con el método de “arrastrar y soltar” parecía demasiado complicado.

   Solución:

   Por simplicidad, se decidió modificar la manera de colocar una cámara en el mapa. Simplemente se desactivó la opción de arrastrar las cámaras del contenedor en donde se encuentran, y éstas ahora se crearían en el mapa al hacer click sobre el mismo.

9. Problema #9:

   En relación con el problema anterior, aún había que resolver cómo pasar la información de las cámaras al fichero QML. La referencia al fichero QML se encuentra en **mainwindow.cpp**, pero es el fichero **dragwidget.cpp** el que gestiona los “clicks” que se hacen sobre los elementos que están en este contenedor.

   Solución:

   Para hacer frente a este problema, se optó por convertir las variables de la clase “MainWindow” **obj** (instancia de QQuickItem que tenía la referencia al objeto raíz de **mapviewer.qml**) y **objTam** (variable que contenía al tamaño del objeto a detectar) en variables estáticas… de esta forma, estas variables se podrían llamar desde cualquier fichero del proyecto de la siguiente forma: **MainWindow::<nombre_de_variable>**, y así, desde la clase **Drag Widget** se podía utilizar **obj** para poder hacer uso del método “setProperty” para mandar información como **objTam** y demás características de la cámara al fichero **mapviewer.qml**.

5 Pruebas y resultados

Este apartado es de los más importantes, pues es aquí donde se realizaron todas las pruebas necesarias para garantizar el correcto funcionamiento de la aplicación, forzando todo tipo de situaciones con el objetivo de detectar posibles errores en la aplicación. Se hicieron diferentes tipos de pruebas, así como pruebas de caja negra, pruebas de caja blanca, pruebas unitarias y de integración[9]. A continuación, vamos a describir detalladamente las fases de pruebas más importantes.

5.1 Pruebas unitarias

Primero se van a realizar pruebas unitarias, ya sea de uno o varios métodos dentro de una misma clase, o de una clase completa.

5.1.1 Registro

Después de haber implementado el registro de nuevos usuarios, había que probar su correcto funcionamiento, por lo que se decidió hacer diversas pruebas de caja negra:

Entrada		**Salida esperada**	**Salida de la aplicación**
Usuario	***Contraseña***		
borch	pepe	El registro se ha realizado correctamente	El registro se ha realizado correctamente
campo vacío	polo	Error al registrarse: No puede dejar campos vacíos	Error al registrarse: No puede dejar campos vacíos
borch	*campo vacío*	Error al registrarse: No puede dejar campos vacíos	Error al registrarse: No puede dejar campos vacíos
borch	algo	Error al registrarse: El usuario ya existe	Error al registrarse: El usuario ya existe

Tabla 1. Pruebas de caja negra para el registro de usuarios

De esta forma, las pruebas unitarias para el registro de nuevos usuarios a la aplicación se concluyen de forma exitosa.

5.1.2 Login

Las pruebas para el inicio de sesión de usuarios ya existentes en la aplicación se realizarán de forma similar a las que se hicieron para el registro de usuarios. Por lo que haremos pruebas de caja negra:

Entrada		**Salida esperada**	**Salida de la aplicación**
Usuario	***Contraseña***		
borch	pepe	Login correcto. Se muestra la interfaz correspondiente al usuario común	Login correcto. Se muestra la interfaz correspondiente al usuario común

borch	*campo vacío*	Error al iniciar sesión: No puede dejar campos vacíos	Error al iniciar sesión: No puede dejar campos vacíos
campo vacío	pepe	Error al iniciar sesión: No puede dejar campos vacíos	Error al iniciar sesión: No puede dejar campos vacíos
birch	pepe	Error al iniciar sesión: Usuario y/o contraseña incorrecta/s	Error al iniciar sesión: Usuario y/o contraseña incorrecta/s
borch	palo	Error al iniciar sesión: Usuario y/o contraseña incorrecta/s	Error al iniciar sesión: Usuario y/o contraseña incorrecta/s
admin	admin	Login correcto. Se muestra la interfaz correspondiente al administrador	Login correcto. Se muestra la interfaz correspondiente al administrador

Tabla 2. Pruebas de caja negra para el inicio de sesión

De esta forma, las pruebas unitarias para el inicio de sesión se concluyen de forma exitosa.

5.1.3 Contenedor de cámaras

Para probar el contenedor de cámaras, el cual es representado por la clase **DragWidget**, se necesitarán realizar pruebas de caja blanca, ya que se deberá probar la lógica interna más que únicamente probar entradas y salidas.

En mayor medida, se ha probado el método "mousePressEvent", el cual es llamado cuando un elemento dentro del **DragWidget** es pulsado. Se podía acceder a las características del objeto pulsado mediante la siguiente línea de código:

```
Camera *child = static_cast<Camera*>(childAt(event->pos()));
```

Cabe destacar que el objeto que interesa seleccionar es de tipo **Camera**, sin embargo, en las primeras pruebas, al pulsar sobre otros elementos que no fuesen la imagen de la cámara, ocurría una violación de segmento y la ejecución del sistema se terminaba. Debuggueando, se detectaron dos escenarios de error:

- El primero es que se pulsaba el nombre de la cámara, y es que éste, a pesar de haberlo considerado en la clase **MainWindow** como un complemento de tipo **Camera** (el cual hereda de **QLabel**), no tenía un **PixMap** definido, por lo que al llegar a la siguiente línea de código, ocurría una violación de segmento:

```
QPixmap pixmap = *child->pixmap();
```

- El otro escenario es que se pulsaba el fondo del **DragWidget** el cual, gracias a debugguear, se observó que varias veces o se devolvía **nullptr** (puntero nulo o vacío), o era un complemento que no era de tipo **Camera**, por lo que, según el caso, o daba violación de segmento cuando se quería acceder a alguna propiedad de un complemento vacío, o pasaba lo mismo que en el primer escenario.

Por esto, tras diversas pruebas, se optó por poner la siguiente condición al principio:

```
if (child == nullptr || child->objectName() != "cameraIcon")
    return;
```

De esta forma, si el nombre del objeto pulsado no es "cameraIcon" (cosa que se define en la clase **MainWindow**), o directamente devuelve un puntero vacío, no se hace nada con el método "mousePressEvent". Se implementó esta misma condición en el método "mouseDoubleClickEvent" de la misma clase para asegurar por completo el correcto funcionamiento del contenedor.
Acto seguido, se volvió a ejecutar la aplicación, se abrió la interfaz para el usuario común, y se intentó forzar el contenedor de cámaras, haciendo click de forma continuada y rápida en cada rincón del contenedor, así como doble clicks, y todo funcionó correctamente, como se esperaba.

5.1.4 Definir una cámara nueva

Al tratarse de un formulario, se van a realizar pruebas de caja negra para comprobar de forma rápida el correcto funcionamiento de esta funcionalidad.

Entrada				**Salida esperada**	**Salida de la aplicación**
Nombre de la cámara	***Tamaño del sensor (pulgadas)***	***Resolución***	***Óptica (m)***		
Cam8	1/3	CIF	0.005	Se ha registrado correctamente	Se ha registrado correctamente
campo vacío	1/3	CIF	0.005	Error al registrar cámara: No puede dejar campos vacíos	Error al registrar cámara: No puede dejar campos vacíos
Cam8	1/3	CIF	*campo vacío*	Error al registrar cámara: No puede dejar campos vacíos	Error al registrar cámara: No puede dejar campos vacíos
Cam8	1/3	CIF	0.006	Error al registrar cámara: La cámara ya existe	Error al registrar cámara: La cámara ya existe

Tabla 3. Pruebas de caja negra para la creación de cámaras nuevas

Al tener las salidas que se esperaban, se puede concluir que las pruebas han sido satisfactorias.

5.1.5 Modificar una cámara existente

Como se van a utilizar los mismos datos de entrada, además de realizar de nuevo pruebas de caja negra para esta funcionalidad, también se van a realizar éstas de forma similar a las de la anterior funcionalidad.

Entrada				**Salida esperada**	**Salida de la aplicación**
Nombre de la cámara	***Tamaño de sensor (pulgadas)***	***Resolución***	***Óptica (m)***		
Cam6	1/2.7	4CIF	0.011	Se ha modificado la cámara correctamente	Se ha modificado la cámara correctamente
campo vacío	1/3	CIF	0.011	Error al modificar cámara: No puede dejar campos vacíos	Error al modificar cámara: No puede dejar campos vacíos
Cam5	1/3	CIF	*campo vacío*	Error al modificar cámara: No puede dejar campos vacíos	Error al modificar cámara: No puede dejar campos vacíos

Tabla 4. Pruebas de caja negra para la modificación de una cámara existente

Puesto que las diferentes salidas de la aplicación para esta funcionalidad coinciden con las salidas esperadas, se concluye que las pruebas se han realizado con éxito.

5.2 Pruebas de integración

Después de haber realizado las pruebas unitarias más importantes, ahora toca realizar las pruebas de integración en las que se probarán prácticamente todas las clases del sistema.

5.2.1 Colocar cámara en el mapa

La prueba de integración más destacada es la de colocar varias cámaras sobre el mapa. Vamos a probar varios escenarios de caja blanca.

5.2.1.1 Se selecciona una cámara, pero no el objeto a detectar

Primero se selecciona una cámara dentro del contenedor de cámaras. Al hacer esto, se accede a la base de datos para extraer los datos que nos interesan de la cámara, los cuales en este caso serán la distancia focal, la anchura y altura del sensor (en metros) y la altura de la resolución (en píxeles). Acto seguido, se pasan estos 5 datos al fichero **mapviewer.qml** (es el que gestiona toda la funcionalidad del mapa) mediante el siguiente fragmento de código de la clase **DragWidget** (más concretamente, del método "mousePressEvent"):

```
MainWindow::obj->setProperty("dist_focal", dist_focal);
MainWindow::obj->setProperty("sens_width", sens_width);
MainWindow::obj->setProperty("sens_height", sens_height);
MainWindow::obj->setProperty("resol_height", resol_height);
```

Habiendo seleccionada la cámara, ahora se pulsa sobre el mapa, pero debajo de éste sale el siguiente mensaje: "Te falta seleccionar el objeto que deseas detectar". Esta es la salida esperada, puesto que es el caso "else", lo cual se traduce a que es la acción a realizar si se tienen valores distintos a '0.0' para la distancia focal, anchura y altura del sensor y altura de resolución, pero aún no hay ningún valor para el objeto que se va a detectar. A continuación, se muestra el trozo de código del fichero **mapviewer.qml** que se encarga de la acción antes explicada:

```
If(dist_focal != 0.0 && sens_height != 0.0 && sens_width != 0.0 && resol_height != 0.0
                                    && objTam != 0.0){
                                    …
}else if(dist_focal == 0.0 || sens_height == 0.0 || sens_width == 0.0 || resol_height == 0.0){
                                    …
                                    }else{
                output.text="Te falta seleccionar el objeto que deseas detectar";
                                    }
```

5.2.1.2 Se selecciona el objeto a detectar, pero no la cámara

En la interfaz del usuario común, además del mapa, contenedor de cámaras, etc., también se dispone de una sección con una lista desplegable para poder seleccionar diversos objetos que se quieren detectar. Al seleccionar uno y pulsar en "Aceptar", se llama al método "acceptObjectToDetectSelection_button", el cual dispone de una variable "objTam" a la que, según el objeto, se le asigna un valor decimal que representará la altura del objeto a detectar, y finalmente se llama a la siguiente línea de código:

```
obj->setProperty("objTam", objTam);
```

Después, al pulsar sobre el mapa, sale el siguiente mensaje debajo de éste: "Te falta seleccionar la cámara que deseas colocar en el mapa". Esto es correcto, pues esta acción está dentro de la condición de cuando tanto la distancia focal como la anchura y altura del sensor y altura de la resolución están a '0.0'. A continuación, se muestra el trozo de código del fichero **mapviewer.qml** que se encarga de gestionar la acción antes explicada:

```
if(dist_focal != 0.0 && sens_height != 0.0 && sens_width != 0.0 && resol_height != 0.0
&& objTam != 0.0){
                …
}else if(dist_focal == 0.0 || sens_height == 0.0 || sens_width == 0.0 ||
resol_height == 0.0){
        output.text="Te falta seleccionar la cámara que deseas colocar en el
mapa";
        }else{
                …
        }
```

5.2.1.3 Si no se selecciona nada, y se pulsa sobre el mapa

En este caso tanto la salida esperada como la que tiene el sistema es igual a la del caso anterior, pues al no tener ni distancia focal, ni anchura ni altura del sensor ni anchura de la resolución, no cumple con la primera condición del fragmento de código que ya se ha puesto veces anteriores, y la siguiente condición es la que cumple, por lo que ese es el mensaje que muestra.

5.2.1.4 Si se selecciona tanto una cámara como un objeto a detector

Al seleccionar la cámara, se manda su información al fichero **mapviewer.qml** de la misma forma que se explicó en el apartado 5.2.1.1., y seleccionar un objeto a detectar y pulsar sobre "Aceptar", se pasa esa información al fichero **mapviewer.qml** de la misma manera explicada en el apartado 5.2.1.2, por lo que la salida esperada debería ser que en el fragmento de código presentado anteriormente se acceda a la primera condición y la cámara se muestre en el mapa junto con su campo de visión, y es exactamente lo que sucede.

Después de probar los 4 posibles escenarios para colocar cámaras en el mapa, se puede concluir que esta funcionalidad cumple de manera correcta y satisfactoria.

6 Conclusiones

Al final del proyecto, se han conseguido solucionar la mayor parte de los problemas a los que me he enfrentado y se han cumplido con los objetivos propuestos desde el principio del proyecto:

- El sistema que dispone de dos perfiles de usuario: un usuario administrador y un usuario común. Mientras que puede haber tantos usuarios comunes se quiera, sólo puede haber UN usuario administrador, el cual ya está añadido en la base de datos. Considero esta decisión acertada, pues así nos aseguramos de que la persona que defina nuevas cámaras las modifique o las elimine tenga conocimientos en las mismas, en sensores, etc., y no cualquiera que pueda crear y borrar cámaras como le plazca.
- Una interfaz gráfica de usuario intuitiva. Qt Creator ha facilitado mucho esta tarea, pues de manera sencilla y cómoda permitió crear no sólo la interfaz gráfica del sistema, sino que también toda la lógica que éste tiene. Cabe destacar que gracias al Qt Designer del que dispone Qt Creator, la mayor parte de la GUI de la aplicación se realizó a base de soltar y arrastrar elementos de la interfaz gráfica; hubo varias partes de la interfaz que se tuvieron que diseñar con código, pues se modificaban de manera dinámica durante la ejecución del sistema.
- Se pueden registrar nuevas cámaras, así como modificarlas. El usuario administrador de manera cómoda puede crear, modificar y eliminar cualquier modelo de cámara, lo cual, a nivel de implementación, se ha conseguido sencillamente mediante una base de datos, accediendo a la misma para crear una cámara nueva, modificar una existente o incluso eliminarla según sea el caso.
- Se puede exportar en una o varias imágenes un proyecto de videovigilancia para el cliente. De manera sencilla se puede exportar la cantidad de imágenes PNG o JPEG que se desee del proyecto, con el nombre y ubicación que el usuario le quiera asignar. Estas imágenes muestran a las cámaras colocadas en el mapa con sus campos de visión.
- Se dispone de un código modular. Las características de la cámara sólo son controladas por la clase **Camera**, la funcionalidad del contenedor de las cámaras existentes sólo es gestionada por la clase **DragWidget**, el mapa únicamente es gestionado por el fichero **mapviewer.qml**, y todo el resto de la funcionalidad, la cual tiene que ver con las diferentes interacciones en la ventana de la aplicación, es gestionada por la clase **MainWindow**.

Cabe destacar la progresiva experiencia que fui adquiriendo tanto con el lenguaje de programación C++ como con el framework Qt. En primer lugar, con C++ dado que, a pesar de no tener experiencia previa con este lenguaje, sí que tengo experiencia con otros lenguajes orientados a objetos como Python y Java, así como con su pariente, C, por lo que C++, a pesar de haber sido un poco complicado al principio, acabó siendo cómodo para poder desarrollar el software con éxito. Algo similar se puede decir de Qt, al principio costó acostumbrarse a esta herramienta, pero a medida que se fue avanzando en los objetivos y se fueron resolviendo los diferentes problemas que se presentaron por el camino, el framework resultaba cada vez más cómodo, en especial a la hora de desarrollar la interfaz gráfica de usuario, tanto haciendo uso de Qt Designer como del propio código.

Sin embargo, considero que el sistema pudo haber sido mucho más usable; por ejemplo, se pudo haber trabajado mucho más el tema de arrastrar una cámara y soltarla en una parte del mapa y que se mostrase la cámara que correspondiese, pero dado el enfoque que le di a la lógica de la mayor parte de las funcionalidades de la aplicación, decidí tomar algunas medidas para solucionar dichos problemas de manera que esas funcionalidades se pudiesen cumplir de la forma más cómoda posible para el usuario.

Durante la elaboración del proyecto, me he topado con varias limitaciones de Qt que no me permitieron realizar algunas funcionalidades de una forma más profesional y cómoda para el usuario. Por ejemplo, según la implementación que seguí para mostrar tanto la cámara como su campo visual como un solo elemento, no me fue posible girar la cámara junto con su campo visual utilizando el ratón, o que no había ningún método o procedimiento dentro de QML para exportar el proyecto a PDF, cosa que me hubiese permitido, además de exportar la imagen, exportar la información de las cámaras utilizadas en el mapa.

Otra de las complicaciones durante el proyecto ha sido el uso de VirtualBox para la ejecución de Linux, sistema operativo en el que desarrollaba la aplicación, pues, al ejecutar la máquina virtual, la máquina anfitriona se ralentizaba muchas veces, llegando en varias ocasiones a congelarse, obligándome a apagar el ordenador forzosamente. Incluso una vez se dañó la máquina virtual, teniendo que volver a instalar todo desde 0 (afortunadamente, con un respaldo del proyecto guardado).

A pesar de estas complicaciones, considero que el uso de VirtualBox me simplificó bastante las cosas, pues no era necesario hacer todo el tedioso proceso de particionar el disco duro, arrancar el ordenador con el USB que contenía la imagen ISO de Ubuntu, etc. Virtualbox también me permitió copiar información de manera sencilla entre la máquina virtual y la máquina anfitriona. Tampoco era necesario apagar el ordenador y volverlo a arrancar para acceder a un sistema operativo determinado, y no me tenía que preocupar si había un fichero importante en otro sistema operativo y necesitaba de su uso.

Del mismo modo, a pesar de las limitaciones que tuve con Qt, considero que ha sido una herramienta bastante útil en este proyecto, pues me ha permitido hacer de manera sencilla muchas cosas que nunca me hubiese imaginado hacer con otro entorno de programación, como por ejemplo colocar cámaras en un mapa y que éste muestre el campo visual de las propias cámaras según sus características, pues este entorno dispone de una basta cantidad de documentación y ejemplos (los cuales se pueden probar y ver su código gracias a que Qt es de código abierto) que el propio Qt Creator pone a disposición de todo el mundo. Además, Qt ha sido esencial a la hora de implementar la interfaz de usuario, pues, comparándolo con AWT/Swing (herramientas de GUI de Java que he utilizado otros años en la carrera), Qt es más intuitivo, por lo que me ha sido sencillo pillar el truco para programar GUIs en Qt de forma rápida y sencilla. Por esto, a pesar de haber tenido la "complicación" de haber empezado lento con la implementación de la aplicación dado el poco conocimiento del *framework*, considero que ha sido un buen acierto haber utilizado éste para la elaboración del sistema, tanto por la parte gráfica como por la parte lógica.

En general, este proyecto ha sido el primer paso para la elaboración de software en un contexto más realista de cara al futuro, dada la organización mental que este proyecto ha requerido y el aprendizaje de nuevas tecnologías que no sólo han ayudado a que analice los diferentes frameworks, lenguajes de programación, etc., y elija el que más se acomoda a mis necesidades (cosa que he tenido que hacer en este proyecto), sino también para que en un futuro sea capaz de realizar el mismo análisis y decidir de manera correcta y firme qué tecnología/s tendrán las mayores ventajas para el desarrollo del proyecto en general.
En cuanto a la aplicación, con el mercado de la videovigilancia y la analítica de vídeo en auge, cualquier persona que busque aumentar la seguridad de su recinto se verá interesada en probar este sistema simple que le ayudará a planear de manera eficaz y lo más barata posible la instalación de un CCTV para su negocio o más importante aún… su hogar.

Referencias

[1] Akshay Upadhyay, “Formula to Find Bearing or Heading angle between two points: Latitude Longitude”, GIS MAP, https://www.igismap.com/formula-to-find-bearing-or-heading-angle-between-two-points-latitude-longitude/

[2] Chris Veness, “Vincenty solutions of geodesics on the ellipsoid”, Movable Type Scripts, https://www.movable-type.co.uk/scripts/latlong-vincenty.html

[3] Computer Hope, “VirtualBox”, Computer Hope, 26 de abril de 2017, https://www.computerhope.com/jargon/v/virtualbox.htm

[4] Floris, “Size of object from its image [duplicate]”, Stack Exchange, 12 de julio de 2015, https://physics.stackexchange.com/questions/193716/size-of-object-from-its-image

[5] HKExnews, “Tamaño del mercado mundial de videovigilancia desde 2009 hasta 2019, por tecnología (en millones de dólares)”, Statista – The Statistics Portal, agosto de 2015, https://es.statista.com/estadisticas/638251/videovigilancia-tamano-del-mercado-en-todo-el-mundo-por-tecnologia/

[6] Madridiario, “Las cámaras de vigilancia son unas de las principales medidas de seguridad de hogares y empresas”, Madridiario, 10 de agosto del 2016, https://www.madridiario.es/436311/camaras-vigilancia-seguridad-hogar

[7] Meritxell M. Pauné, “¿Las videocámaras reducen la delincuencia?”, La Vanguardia, 6 de abril de 2011, https://www.lavanguardia.com/vida/20110406/54136649079/las-videocamaras-reducen-la-delincuencia.html

[8] Ricardo Carrillo de Albornoz, “Tamaño y Resolución del Sensor”, todo-fotografía, 2011, https://todo-fotografia.com/tecnica/tamano-y-resolucion-del-sensor/

[9] Sten Pittet, “The different types of software testing”, Atlassian, https://es.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

[10] “About Qt”, Qt Wiki, 26 de abril de 2019, https://wiki.qt.io/About_Qt

[11] “About SQLite”, SQLite, https://www.sqlite.org/about.html

[12] “Analítica de vídeo”, Axis Communications, https://www.axis.com/es-es/learning/web-articles/video-analytics/what-are-video-analytics

[13] “CCTV Video Broadcast Standards (NTSC/PAL/SECAM)”, Discount Security Cameras, 27 de junio de 2015, http://www.discount-security-cameras.net/cctv-video-resolutions.aspx

[14] “Draggable Icons Example”, Qt Documentation, 2018, https://doc.qt.io/qt-5.11/qtwidgets-draganddrop-draggableicons-example.html

[15] “Executing SQL Statements”, Qt Documentation, 2019, https://doc.qt.io/qt-5/sql-sqlstatements.html

[16] “Image Formation by Lenses”, Lumen Learning, https://courses.lumenlearning.com/physics/chapter/25-6-image-formation-by-lenses/

[17] “Image Sensor Format”, Spot Imaging, 2019, http://www.spotimaging.com/resources/glossary/image-sensor-format/

[18] “ListModel QML Type”, Qt Documentation, 2019, https://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html

[19] “MapPolygon QML Type”, Qt Documentation Snapshots, 2019, https://doc-snapshots.qt.io/qt5-dev/qml-qtlocation-mappolygon.html

[20] “MapQuickItem QML Type”, Qt Documentation, 2019, https://doc.qt.io/qt-5/qml-qtlocation-mapquickitem.html

[21] "Map Viewer (QML), Qt Documentation Snapshots, 2019, https://doc-snapshots.qt.io/qt5-5.9/qtlocation-mapviewer-example.html
[22] "MouseArea QML Type", Qt Documentation, 2019, https://doc.qt.io/qt-5/qml-qtquick-mousearea.html
[23] "Property Binding", Qt Documentation Archives, 2016, https://doc.qt.io/archives/qt-4.8/propertybinding.html
[24] "QFormLayout Class", Qt Documentation Archives, 2016, https://doc.qt.io/archives/qt-4.8/qformlayout.html
[25] "QML Mouse Events", Qt Documentation Archives, 2016, https://doc.qt.io/archives/qt-4.8/mouseevents.html
[26] "QQuickItem Class", Qt Documentation, 2019, https://doc.qt.io/qt-5/qquickitem.html
[27] "QQuickWidget Class", Qt Documentation, 2019, https://doc.qt.io/qt-5/qquickwidget.html
[28] "QScrollArea Class", Qt Documentation Archives, 2017, https://doc.qt.io/archives/qt-5.6/qscrollarea.html
[29] "QStackedWidget Class", Qt Documentation, 2019, https://doc.qt.io/qt-5/qstackedwidget.html
[30] "Qt History", Qt Wiki, 24 de noviembre de 2018, https://wiki.qt.io/Qt_History
[31] "QWidget Class", Qt Documentation, 2019, https://doc.qt.io/qt-5/qwidget.html
[32] "Rotation", Qt Documentation, 2019, https://doc.qt.io/qt-5/qml-qtquick-rotation.html
[33] "Sensor Size & Field of View", Panavision, 2015, http://www.panavision.com/sites/default/files/docs/documentLibrary/2%20Sensor%20Size%20FOV%20(2).pdf

Glosario

COMBOBOX	Elemento de la interfaz de usuario que permite desplegar una lista de elementos existentes.
BINDING	"Ligadura" o referencia a otro símbolo más largo y complicado, y que se usa frecuentemente.
ACID	Características de los parámetros que permiten clasificar las transacciones de los sistemas de gestión de bases de datos. Acrónimo de *Atomicity, Consistency, Isolation and Durability*.
FRAMEWORK	Entorno de desarrollo.
GUI	Siglas de *Graphic User Interface* (Interfaz gráfica de usuario).
WIDGET	En el contexto de interfaz gráfica de usuario, se trata de un componente o control visual que el programador reutiliza.
LAYOUT	El arreglo de varios elementos visuales en una página.
CARDLAYOUT	Layout que gestiona los diferentes componentes visuales de tal manera que sólo uno esté visible a la vez.
LABEL	Etiqueta.
LOGIN	Inicio de sesión.
SCROLL	Desplazamiento en 2D de los contenidos que conforman un contenedor.
BOOLEANO	Que sólo acepta como valores "true" (verdadero) o "false" (falso).
BEARING	Ángulo horizontal entre la dirección de un objeto y otro, o entre el objeto y el verdadero norte (por ejemplo, un objeto al norte tendría un "bearing" de 0 grados, al este tendría 90 grados, etc…).
API	Siglas para Application Programming Interface (Interfaz de Programación de Aplicaciones). Es un conjunto de subrutinas, funciones y procedimientos que ofrece una determinada biblioteca.
PRUEBAS UNITARIAS	Consisten en pruebas que se realizan sobre métodos y funciones dentro de una clase individual.

PRUEBAS DE INTEGRACIÓN	Consisten en pruebas que se realizan sobre el conjunto parcial o total de diferentes módulos que están relacionados entre sí.
DEBUGGEAR	Es la acción de depurar el código. Esto es, ejecutar el programa de forma controlada siguiendo cada instrucción con el fin de localizar “bugs”, errores de ejecución, etc.
CCTV	Acrónimo para Circuito Cerrado de Televisión (en inglés, Closed Circuit Television). Es un circuito de cámaras privado, es decir, está pensado para un número recudido de espectadores, a diferencia de la televisión convencional